HILOS DE EJECUCIÓN EN JAVA

1. Hilos

Un hilo es un flujo de control dentro de un programa. Creando varios hilos podremos realizar varias tareas simultáneamente. Cada hilo tendrá sólo un contexto de ejecución (contador de programa, pila de ejecución). Es decir, a diferencia de los procesos UNIX, no tienen su propio espacio de memoria sino que acceden todos al mismo espacio de memoria común, por lo que será importante su sincronización cuando tengamos varios hilos accediendo a los mismos objetos.

1.1. Creación de hilos: La clase Thread y la interface Runnable

En Java los hilos están encapsulados en la clase **Thread**. Para crear un hilo tenemos dos posibilidades:

- Heredar de **Thread** redefiniendo el método *run()*.
- Crear una clase que implemente la interfaz **Runnable** que nos obliga a definir el método *run()*.

En ambos casos debemos definir un método *run()* que será el que contenga el código del hilo. Desde dentro de este método podremos llamar a cualquier otro método de cualquier objeto, pero este método *run()* será el método que se invoque cuando iniciemos la ejecución de un hilo. El hilo terminará su ejecución cuando termine de ejecutarse este método *run()*. Existen otras formas de terminación forzada, que se verán más adelante.

Para crear nuestro hilo mediante herencia haremos lo siguiente:

```
public class EjemploHilo extends Thread
{
    public void run() {
        // Código del hilo
    }
}
```

Una vez definida la clase de nuestro hilo deberemos instanciarlo y ejecutarlo de la siguiente forma:

```
Thread t = new EjemploHilo();
t.start();
```

Crear un hilo heredando de **Thread** tiene el problema de que al no haber herencia múltiple en Java, si heredamos de **Thread** no podremos heredar de ninguna otra clase, y por lo tanto un hilo no podría heredar de ninguna otra clase.

Este problema desaparece si utilizamos la interfaz **Runnable** para crear el hilo, ya que una clase puede implementar varios interfaces. Definiremos la clase que contenga el hilo como se muestra a continuación:

```
public class EjemploHilo implements Runnable
{
    public void run() {
        // Código del hilo
    }
}
```

Para instanciar y ejecutar un hilo de este tipo deberemos hacer lo siguiente:

```
Thread t = new Thread(new EjemploHilo());
t.start();
```

Esto es así debido a que en este caso **EjemploHilo** no deriva de una clase **Thread**, por lo que no se puede considerar un hilo, lo único que estamos haciendo implementando la interfaz es asegurar que vamos a tener definido el método *run()*. Con esto lo que haremos será proporcionar esta clase al constructor de la clase **Thread**, para que el objeto **Thread** que creemos llame al método *run()* de la clase que hemos definido al iniciarse la ejecución del hilo, ya que implementando la interfaz le aseguramos que esta función existe.

1.2. Estado, transiciones, prioridades y propiedades de los hilos

Un hilo pasará por varios estados durante su ciclo de vida.

```
Thread t = new Thread(this);
```

Una vez se ha instanciado el objeto del hilo, diremos que está en estado de *Nuevo hilo*.

```
t.start();
```

Cuando invoquemos su método *start()* el hilo pasará a ser un hilo *vivo*, comenzándose a ejecutar su método *run()*. Una vez haya salido de este método pasará a ser un hilo *muerto*.

La única forma de parar un hilo es hacer que salga del método *run()* de forma natural. Podremos conseguir esto haciendo que se cumpla la condición de salida del bucle principal definido dentro del *run()*. Las funciones para parar, pausar y reanudar hilos estás desaprobadas en las versiones actuales de Java.

Mientras el hilo esté *vivo*, podrá encontrarse en dos estados: *Ejecutable* y *No ejecutable*. El hilo pasará de *Ejecutable* a *No ejecutable* en los siguientes casos:

- Cuando se encuentre dormido por haberse llamado al método *sleep()*, permanecerá *No ejecutable* hasta haber transcurrido el número de milisegundos especificados.
- Cuando se encuentre bloqueado en una llamada al método *wait()* esperando que otro hilo lo desbloquee llamando a *notify()* o *notifyAll()*.
- Cuando se encuentre bloqueado en una petición de E/S, hasta que se complete la operación de E/S.

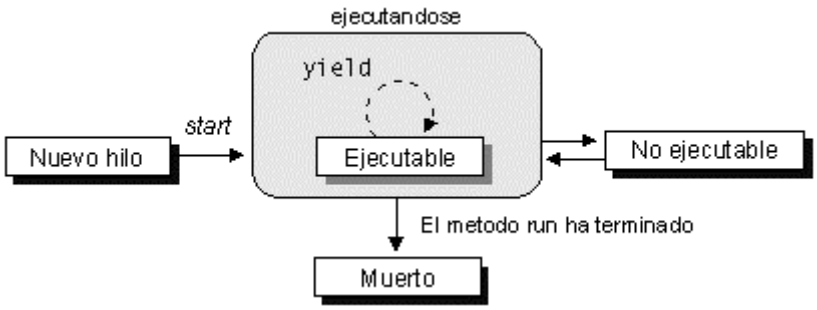

Figura 1. Ciclo de vida de los hilos

Lo único que podremos saber es si un hilo se encuentra vivo o no, llamando a su método *isAlive()*.

Además, una propiedad importante de los hilos será su prioridad. Mientras el hilo se encuentre vivo, el *scheduler* de la máquina virtual Java le asignará o lo sacará de la CPU, coordinando así el uso de la CPU por parte de todos los hilos activos basándose en su prioridad. Se puede forzar la salida de un hilo de la CPU llamando a su método *yield()*. También se sacará un hilo de la CPU cuando un hilo de mayor prioridad se haga *Ejecutable*, o cuando el tiempo que se le haya asignado expire.

Los hilos siempre se ejecutan con una prioridad entre 1 y 10.

Un hilo toma como prioridad por defecto la prioridad del hilo de ejecución que lo crea. La prioridad por defecto es 5.

Podemos modificar la prioridad de un hilo (desde el valor mínimo 1 hasta el máximo 10) mediante el método *setPriority().*

El método *getPriority()* investiga la prioridad de ejecución de un hilo.

Algunas máquinas virtuales no son capaces de reconocer 10 valores diferentes de prioridad, lo que causa un problema al momento de la ejecución, debido a ello, la clase Thread tiene 3 constantes (variables estáticas y finales) que definen un rango de prioridades de hilo.

- Thread.MIN_PRIORITY(1)
- Thread.NORM_PRIORITY(5)
- Thread.MAX_PRIORITY(10)

1.3. Comunicación y sincronización de hilos

Muchas veces los hilos deberán trabajar de forma coordinada, por lo que es necesario un mecanismo de sincronización entre ellos.

Un primer mecanismo de comunicación es la variable cerrojo incluida en todo objeto **Object**, que permitirá evitar que más de un hilo entre en la sección crítica. La sección crítica es la porción de código de un programa en la cual se accede a un recurso compartido (estructura de datos o dispositivo) que no debe ser accedido por más de un hilo en ejecución. Los métodos declarados como *synchronized* utilizan el cerrojo de la clase a la que pertenecen evitando que más de un hilo entre en ellos al mismo tiempo.

```
public synchronized void seccion_critica()
```

```
{
    // Código sección crítica
}
```

También podemos utilizar cualquier otro objeto para la sincronización dentro de nuestro método de la siguiente forma:

```
synchronized (objeto_con_cerrojo)
{
    // Código sección crítica
}
```

Además podemos hacer que un hilo quede bloqueado a la espera de que otro hilo lo desbloquee cuando suceda un determinado evento. Para bloquear un hilo usaremos la función *wait()*, para lo cual el hilo que llama a esta función debe estar en posesión del monitor, cosa que ocurre dentro de un método *synchronized*, por lo que sólo podremos bloquear a un proceso dentro de estos métodos.

Para desbloquear a los hilos que haya bloqueados se utilizará *notifyAll()*, o bien *notify()* para desbloquear sólo uno de ellos aleatoriamente. Para invocar estos métodos ocurrirá lo mismo, el hilo deberá estar en posesión del monitor.

Cuando un hilo queda bloqueado liberará el cerrojo para que otro hilo pueda entrar en la sección crítica y desbloquearlo.

Por último, puede ser necesario esperar a que un determinado hilo haya finalizado su tarea para continuar. Esto lo podremos hacer llamando al método *join()* de dicho hilo, que nos bloqueará hasta que el hilo haya finalizado.

1.4. Grupo de hilos

Los grupos de hilos nos permitirán crear una serie de hilos y manejarlos todos a la vez como un único objeto. Si al crear un hilo no se especifica ningún grupo de hilos, el hilo creado pertenecerá al grupo de hilos por defecto.

Podemos crearnos nuestro propio grupo de hilos instanciando un objeto de la clase **ThreadGroup**. Para crear hilos dentro de este grupo deberemos pasar este grupo al constructor de los hilos que creemos.

```
ThreadGroup grupo = new ThreadGroup("Grupo de hilos");
Thread t = new Thread(grupo,new EjemploHilo());
```

1.5. Resumen de la interfaz de programación (API) de hilos

1.5. 1. Constructores de la clase Thread

Existen varios constructores de la clase Thread (y transferido por herencia a todas sus extensiones)

Desde el punto de vista estructural existen **dos variantes básicas**:

Las que requieren que el código del método run() se especifique explícitamente en la declaración de la clase: Por ejemplo: **Thread** (String *threadName*)

Las que requieren un parámetro de inicialización que implemente la interfaz Runnable: **Thread** (Runnable *threadOb*)

Los restantes constructores resultan de si se asigna un ***nombre al hilo***, y que solo afecta para inicializar ese atributo en la instancia del objeto, y puede utilizarse para que en fases de verificación cada hilo pueda autoidentificarse, o de si se le crea dentro de un ***ThreadGroup***, lo cual limita su accesibilidad y su capacidad de interacción con hilos que han sido creados en otros ThreadGroup.

- **Thread()**
- **Thread(Runnable *ThreadOb*)**
- **Thread(Runnable *ThreadOb*,** String ***ThreadName*)**
- **Thread(String *ThreadName*)**
- **Thread(ThreadGroup *groupOb*,** Runnable ***ThreadOb*)**
- **Thread(ThreadGroup *groupOb*,** Runnable ***ThreadOb*,** String ***ThreadName*)**
- **Thread(ThreadGroup *GroupOb*,** String ***ThreadName*)**

1.5.2. Métodos de clase

Estos son los métodos estáticos que deben llamarse de manera directa en la clase **Thread**.

- ***currentThread()***: Este método devuelve el objeto **thread** que representa al hilo de ejecución que se está ejecutando actualmente.
- ***yield()***: Este método hace que el intérprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible. Es una manera de asegurar que nos hilos de menor prioridad no sufran inanición.

- ***sleep(long)***: El método *sleep()* provoca que el intérprete ponga al hilo en curso a dormir durante el número de milisegundos que se indiquen en el parámetro de invocación. Una vez transcurridos esos milisegundos, dicho hilo volverá a estar disponible para su ejecución. Los relojes asociados a la mayor parte de los intérpretes de Java no serán capaces de obtener precisiones mayores de 10 milisegundos, por mucho que se permita indicar hasta nanosegundos en la llamada alternativa a este método.

1.5.3. Métodos de instancia

- ***start()***: Este método indica al intérprete de Java que cree un contexto del hilo del sistema y comience a ejecutarlo. A continuación, el método *run()* de este hilo será invocado en el nuevo contexto del hilo. Hay que tener precaución de no llamar al método *start()* más de una vez sobre un hilo determinado.
- ***run()***: El método *run()* constituye el cuerpo de un hilo en ejecución. Este es el único método del interfaz**Runnable**. Es llamado por el método *start()* después de que el hilo apropiado del sistema se haya inicializado. Siempre que el método *run()* devuelva el control, el hilo actual se detendrá.
- ***stop()***: Este método provoca que el hilo se detenga de manera inmediata. A menudo constituye una manera brusca de detener un hilo, especialmente si este método se ejecuta sobre el hilo en curso. En tal caso, la línea inmediatamente posterior a la llamada al método *stop()* no llega a ejecutarse jamás, pues el contexto del hilo muere antes de que *stop()* devuelva el control. Una forma más elegante de detener un hilo es utilizar alguna variable que ocasione que el método *run()* termine de manera ordenada. En realidad, **nunca se debería recurrir al uso de este método.**
- ***suspend()***: El método *suspend()* es distinto de *stop()*. *suspend()* toma el hilo y provoca que se detenga su ejecución sin destruir el hilo de sistema subyacente, ni el estado del hilo anteriormente en ejecución. Si la ejecución de un hilo se suspende, puede llamarse a *resume()* sobre el mismo hilo para lograr que vuelva a ejecutarse de nuevo.
- ***setPriority(int)***: El método *setPriority()* asigna al hilo la prioridad indicada por el valor pasado como parámetro. Hay bastantes constantes predefinidas para la prioridad, definidas en la clase **Thread**, tales como`MIN_PRIORITY`, `NORM_PRIORITY` y `MAX_PRIORITY`, que toman los valores 1, 5 y 10, respectivamente. Como guía aproximada de utilización, se puede establecer que la mayor parte de los procesos a nivel de usuario deberían tomar una prioridad en torno a `NORM_PRIORITY`. Las tareas en segundo plano, como una entrada/salida a red o el nuevo dibujo de la pantalla, deberían tener una prioridad cercana a`MIN_PRIORITY`. Con las tareas a las que se fije la máxima prioridad, en torno a `MAX_PRIORITY`, hay que ser especialmente cuidadosos, porque si no se hacen llamadas a *sleep()* o *yield()*, se puede provocar que el intérprete Java quede totalmente fuera de control.
- ***getPriority()***: Este método devuelve la prioridad del hilo de ejecución en curso, que es un valor comprendido entre uno y diez.
- ***setName(String)***: Este método permite identificar al hilo con un nombre menmónico. De esta manera se facilita la depuración de programas multihilo. El nombre mnemónico aparecerá en todas las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.

- ***getName()***: Este método devuelve el valor actual, de tipo cadena, asignado como nombre al hilo en ejecución mediante *setName()*.